Issue
25

Date
JUN
81

DATA CHIP

The Newsletter of the Compucolor Users' Group
Of Rochester, New York
Our 3rd Year

```
10 INPUT "SCALE? ";F
20 XX = 64: YY = 64
30 FOR X = 0 TO 2000
40 A = F * X: XX = XX + SIN(X) * A: YY = YY + COS(X) * A
50 PLOT 6,(X AND 3) + 1,2,XX,YY,255
60 NEXT
15 PLOT 12,3,64,0
```

Editor's notes

The Second Compucolor Seminar was held on May 16th in Rochester, and provided an informative afternoon for some 30-40 Compucolor owners. The first session dealt with basic graphic concepts,a subject with which the lecturer, Tom Bowllan, is well-aquainted. His notes are included in this issue, and should make a valuable reference for those who were there, and interesting reading for those who were not. The second session, by Dale Dewey, dealt with tuning up the CCII color and disk. It was great to see twenty CCII's sitting around the tables in pieces. We learned how to field strip the CCII, purify and converge it, clean its disk, and picked up many little tips from the leaders and others that were well worth the price of admission. If you're interested in Dale's notes, write to Datachip and we'll examine the options of sending them or publishing them.

One tip from the seminar - - Take an old disk cover (the paper package), slit the top of it so that it's closed on two sides only, and afix it to the side of the CCII next to the drive to hold a disk as you swap from one to another during dup's or other operations where you must switch from one disk to another.

In this issue we introduce a new author, Rick Taubold, a CCII newcomer who is very interested in programming and is assisting the club in the production of Datachip with low cost duplicating. Thanks, Rick.

This issue also contains the fifth in the CompuPEEKing series by Jim Minor, and a beautiful chart from Wally Rust. Read on!

Editor
Ben Barlow
161 Brookside Dr.
Rochester, NY 14618

Librarian
Dave Suits
General Studies
R. I. T.
Rochester, NY 14623

Subscriptions
Gene Bailey
28 Dogwood Glen
Rochester, NY 14625

DRAWING SIMPLE GRAPHS

Tom Bowllan 2nd Semi-Annual CHIP Seminar May 16, 1981

Well, a day late and a dollar short, here they are--the notes that were supposed to go with the graphic lecture. Graph drawing can be quite simple on the CCII once you know the codes and a few simple routines. We spent most of our allotted time at the seminar (and a little extra) going over the more commonly used plot codes. If there is any interest in exploring the topic in greater detail, we could possibly continue at some future seminar. If there isn't, then there probably isn't any interest in these notes either--but here they are anyway. My thanks to all involved.

I. Preliminary Planning --- Determine:

A. The purpose or "message" of the graph

B. How input will be entered (formula, file, array, etc.)

C. What output should be shown -- Keep it SIMPLE !

II. Formatting the Output --- Determine:

A. How much "screen-room" the graph will cover

B. Where the graph will be placed on the screen

C. What color combinations will be most effective

D. Type of graph -- points, connected points, V or H bars

E. Type of scaling to be used -- linear, log, percentage, etc.

F. Spacing and indentification along each axis

G. What text and/or numerical data should be displayed

III. Actual Construction

A. Begin with a "clean" non-scrolling screen

B. Find the maximum (and minimum) values

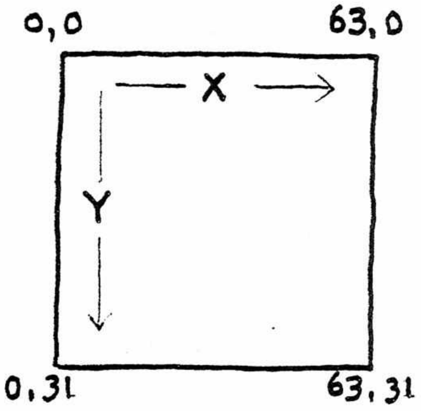
C. Identify the display

D. Draw the axis and (if desired) print scaling lines

E. Print legend, numerical data, and misc. information

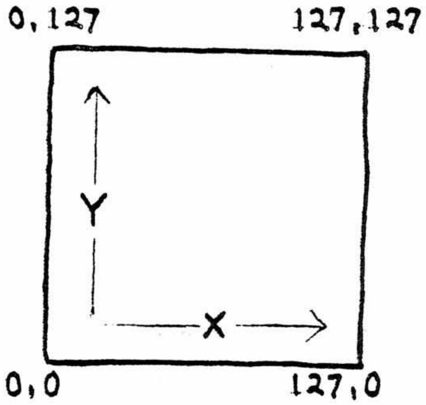
F. "Freeze" the display with a "null-input prompt"

CHARACTER MODE



The CCII screen is divided into a grid of 32 rows and 64 columns. Any position on the screen can be easily reached by using a simple statement: PLOT 3,X,Y , where: X determines the horizontal position (column) and Y determines the vertical position (row). The "origin" is found in the TOP-LEFT corner of the screen at X=0, Y=0. As X increases from 0 to 63, placement is shifted from left to right. As Y increases from 0 to 31, placement is shifted from top to bottom.

PLOT MODE



The screen can also be divided into a grid w/higher resolution: (128 high by 128 wide), which is more suitable for drawing graphs. Following a "PLOT 2", coordinates are still defined by X,Y coordinates, but there are some important differences to be noted: The "origin" is now in the BOTTOM-LEFT corner of the screen at X=0,Y=0. As Y increases from 0 to 127, placement is shifted from bottom to top.

HORIZONTAL COORDINATE CONVERSION

Determining the PLOT MODE horizontal point coordinates is quite simple:

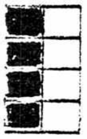
A. In both modes, positions are numbered from left to right.
B. Each character position is divided into just 2 increments.

XC == "X" Character coordinate

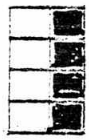
XP == "X" Plot coordinate

LEFT Position

XP = 2 * XC



RIGHT Position



XP = 2 * XC + 1

YC	TOP	UP-MD	LO-MD	BTM
0	127	126	125	124
1	123	122	121	120
2	119	118	117	116
3	115	114	113	112
4	111	110	109	108
5	107	106	105	104
6	103	102	101	100
7	99	98	97	96
8	95	94	93	92
9	91	90	89	88
10	87	86	85	84
11	83	82	81	80
12	79	78	77	76
13	75	74	73	72
14	71	70	69	68
15	67	66	65	64
16	63	62	61	60
17	59	58	57	56
18	55	54	53	52
19	51	50	49	48
20	47	46	45	44
21	43	42	41	40
22	39	38	37	36
23	35	34	33	32
24	31	30	29	28
25	27	26	25	24
26	23	22	21	20
27	19	18	17	16
28	15	14	13	12
29	11	10	9	8
30	7	6	5	4
31	3	2	1	0

VERTICAL COORDINATE CONVERSIONS

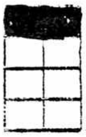
When using the PLOT MODE, each character position is divided into four vertical increments:

1. Top position
2. Upper-middle position
3. Lower-middle position
4. Bottom position

YC == "Y" Character coordinate

YP == "Y" Plot coordinate

TOP Position



YP = 127 - 4 * YC

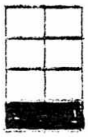
UPPER-MIDDLE Position

YP = 126 - 4 * YC

LOWER-MIDDLE Position

YP = 125 - 4 * YC

BOTTOM Position



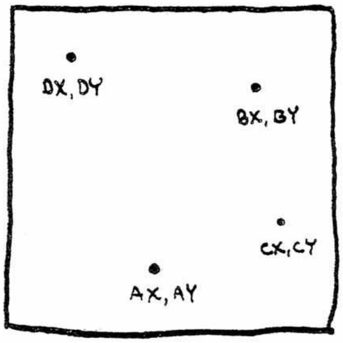
YP = 124 - 4 * YC

Tom Bowllan May 16, 1981

FREQUENTLY USED PLOT MODES

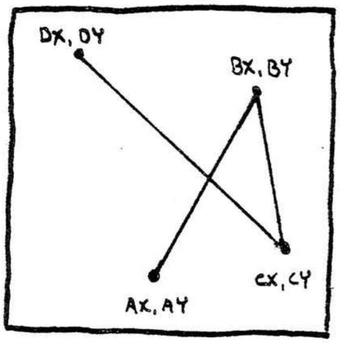
This is a very brief description of some of the PLOT commands and their effect which will be used in construction of some simple graphic displays.

PLOT 2 -- GENERAL PLOT MODE INTRODUCTION



As soon as your CCII receives the "PLOT 2" instruction, it is ready to start drawing something for you. If another PLOT command greater than 239 is entered, it will shift into a more specific plot mode. Several of these are outlined below. Otherwise, the computer will interpret each successive pair of numbers found in all ensuing PLOT statements as "X" and "Y" coordinates of some point, which it will then map on the screen. For example: PLOT 2,AX,AY,BX,BY,CX,CY.

PLOT 242 -- CONNECT THE DOTS

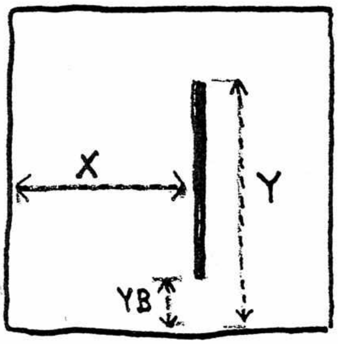


You may wish to connect certain points as you plot them. The computer must first be given the coordinates of a starting point. Follow this with a "PLOT 242" and you'll be set to draw lines in any direction that you want--up, down, right, left or diagonal. Provide more coordinates and the line will be continued to the next point. The "242" need be entered only once; e.g. PLOT 2,AX,AY,242,BX,BY,CX,CY.

PLOT 255 -- STOP THE PLOTTING

Sooner or later, you may decide that there's more to life than the plot Mode--and that you want out! This command is your key to the escape hatch. Use it at the end of each plotting or graphic sequence.

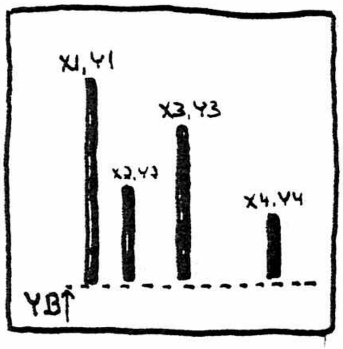
PLOT 246 -- VERTICAL LINES



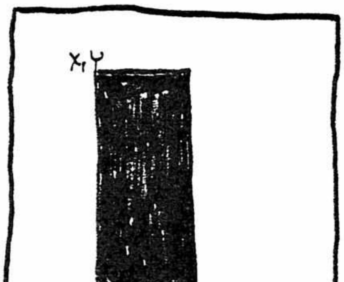
This code enables you to quickly construct vertical lines anywhere on the screen. The next three values that the computer receives determine the line's placement and length.

YB == "Y" coordinate of the Bottom endpoint
X == "X" coordinate (distance fr left margin)
Y == "Y" coordinate of the upper endpoint.

To draw the line, simply: PLOT 2,246,YB,X,Y

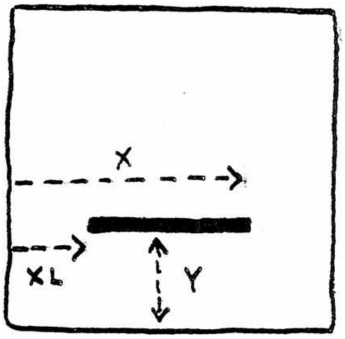


Once "YB" has been entered, any number of additional vertical lines can be drawn from the same base. Neither the "246" nor the "YB" value need be repeated. For example: PLOT 2,246,YB,X1,Y1,X2,Y2,X3,Y3,X4,Y4 etc.



Sometimes it may be desirable to make the line "a little thicker". As noted above, you could shift the "X" value by 1 unit and re-enter the "Y" value; for example: PLOT 2,246,YB,X,Y,X+1,Y,X+2,Y,X+3,Y etc. However these coordinates may be written as a much longer variable expression, and this would require many unnecessary key-strokes and time. Yes, there is a simpler way to accomplish this. Follow the "Y" value with "**PLOT 243**" and then as many "0's" as there are shifts desired. Ex: PLOT 2,246,YB,X,Y,243,0,0,0,0,0,0,0,0

PLOT 250 -- HORIZONTAL LINES



This code works much the same as the vertical bar code. Following the "250", the line is defined by the next three values received:

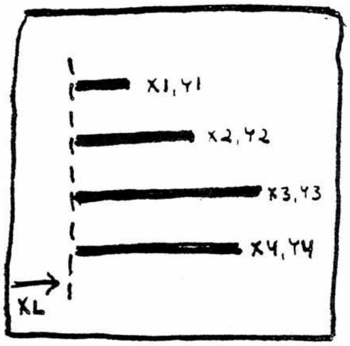
XL == "X" coordinate of the Left endpoint

Y == "Y" coordinate (distance fr bottom)

X == "X" coordinate of the right endpoint.

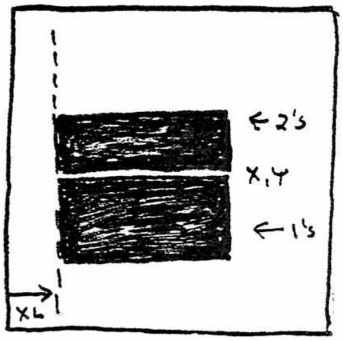
Note that the line must always be drawn from left to right and that when entering the endpoints of the line, "Y" preceeds "X".

It is written: PLOT 2,250,XL,Y,X



Additional horizontal lines can also be drawn from the same base without repeating "250,XL".

Ex: PLOT 2,250,XL,Y1,X1,Y2,X2,Y3,X3 etc.



Thickening the line into a bar is easily done by the use of "**PLOT 247**" after the parameters of the original line have been given. To widen in an upward direction, use a series of "2's". If you prefer to widen toward the bottom instead, use "1's". Here are a couple of examples:

PLOT 2,250,XL,Y,X,247,2,2,2,2,2,2,2,2 etc.

or

PLOT 2,250,XL,Y,X,247,1,1,1,1,1,1,1,1 etc.

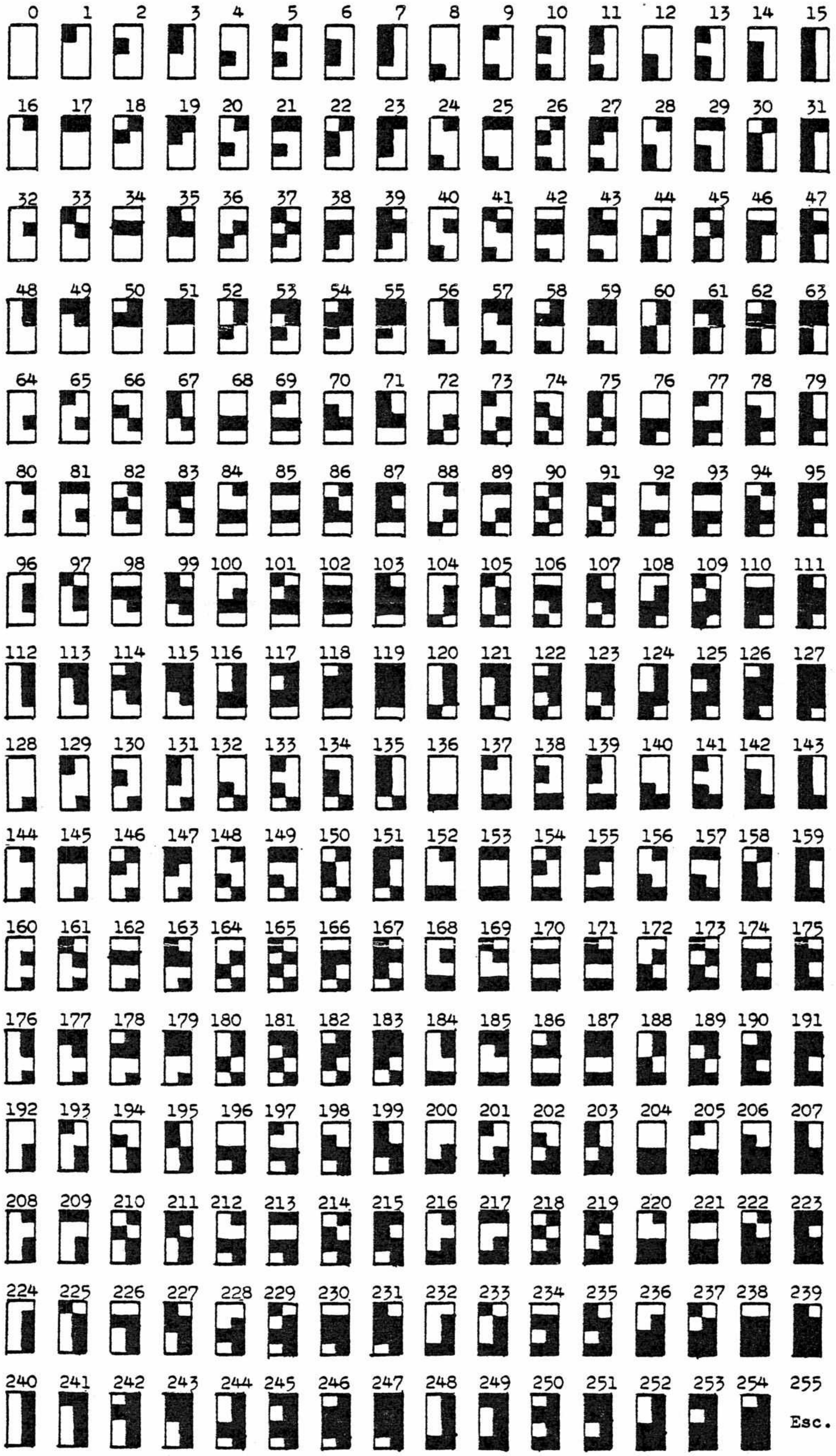
CHARACTERS AVAILABLE ON THE COMPUCOLOR COMPUTER USING GRAPHICS CHARACTER MODE

In each character shown below, black areas represent the foreground color and white areas represent the background color.

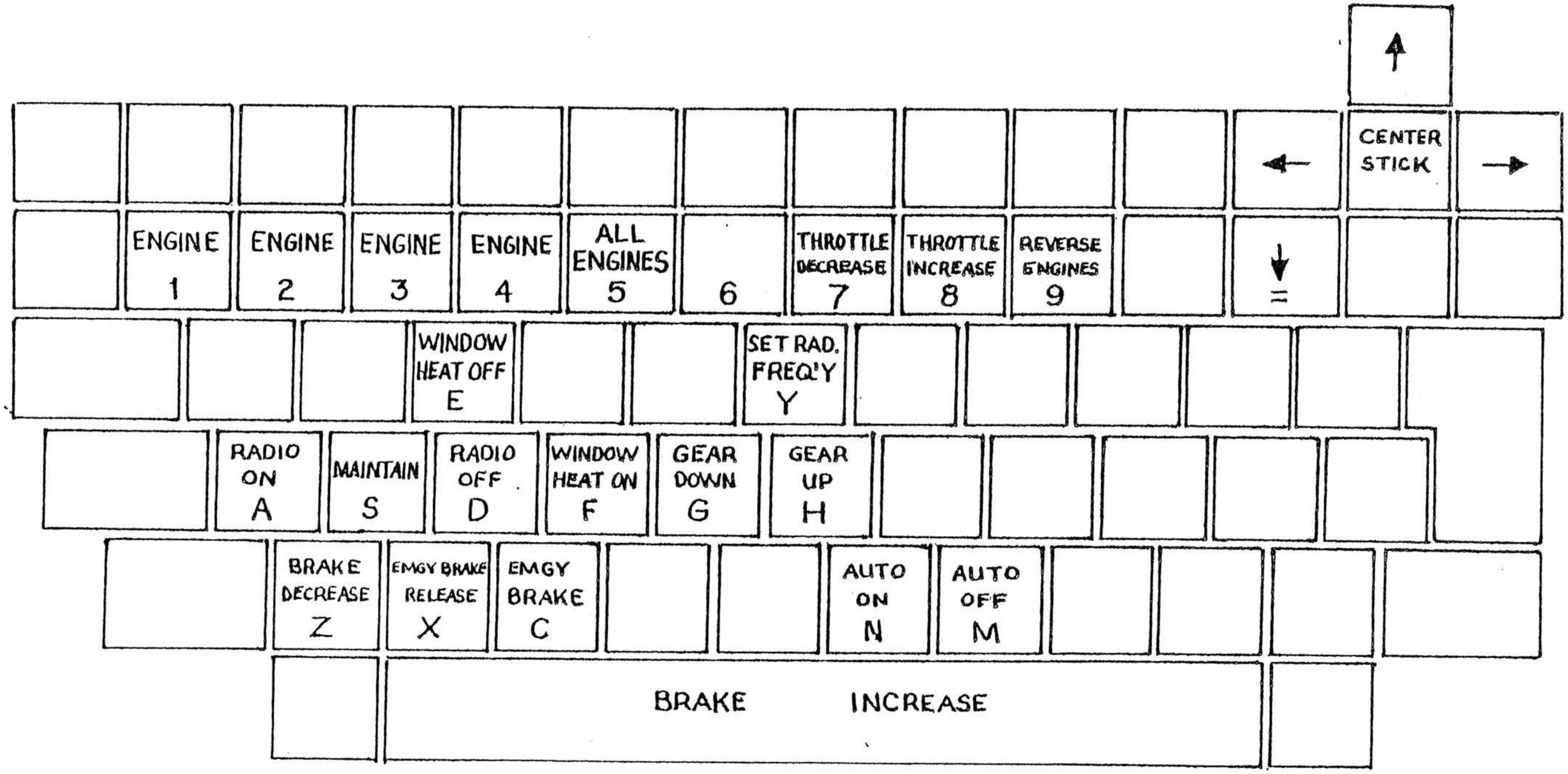
To obtain a string of characters N1, N2,... on the screen, use:

PLOT 2,254,N1,N2,...,255

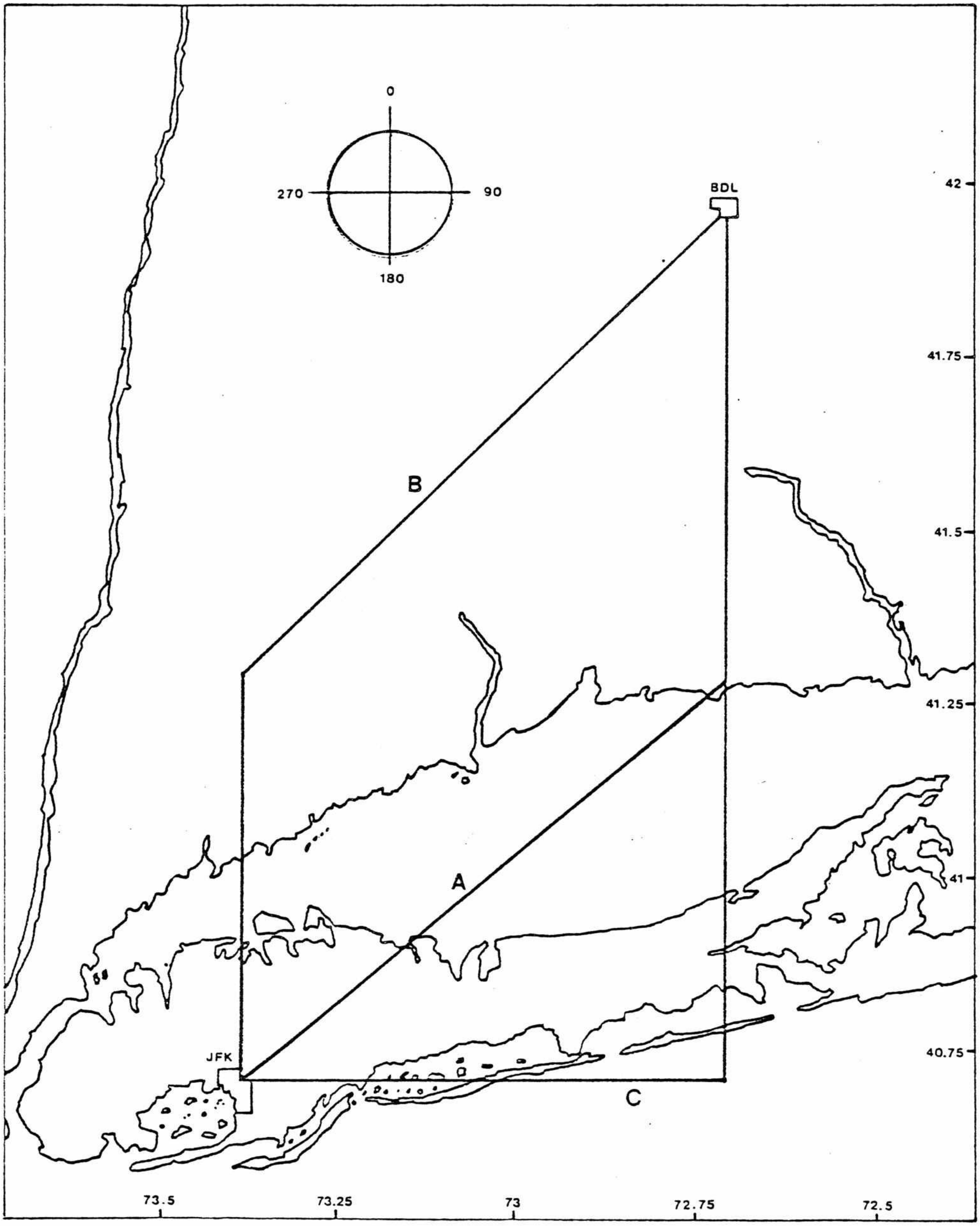
(Wallace R. Rust, May 1981)



One of the most popular programs in our library has been the flight simulation program by R.B. Holley. Here's a chart of the keyboard functions as defined by the program, and a map showing the airports and the three flight plans.
Happy landings!



Keyboard Layout For "Flight Simulation" CHIP Library Program #136, Disk: AW



Map for "Flight Simulation" CHIP Library Program #136, Disk: AW

RANDOM THOUGHTS

by

Rick Taubold

This article begins what I hope will be a regular offering in the Data Chip. In the months to come I plan to cover a wide variety of topics with a special emphasis on algorithms and programming techniques that can be applied to a variety of languages besides BASIC.

First of all let me introduce myself properly. I have been a member of the User's Group for just a few months and have had a CompuColor for about two months (as of this writing). By training I am a Ph.D. biochemist employed at the University of Rochester Medical Center, Dept. of Pediatrics. My interest in computers goes back a way. As a result I feel qualified to speak on the subject of programming. Much of what I share will be scientifically and mathematically oriented, but not always. I would appreciate your comments, criticisms and even suggested topics. You may write to me at: 197 Hollybrook Road, Rochester, New York 14623. So on with the first topic.

"OF DICE AND MEN"

Have you ever wondered how your BASIC language generates its random numbers? Does it work as well as it should and what guarantees that the numbers are random? No, you say? Well, let's take an excursion down the path of random numbers.

Having only had my CompuColor a short while I have not had sufficient time to explore the BASIC ROM to discover what method it uses for random numbers. Most of the random number generators extant in computers today use some variation of the method I will describe below.

It may come as a surprise to most that a random number generator is anything but random. In fact, if it were, it probably would not work very well. The truth is that it behaves in a very predictable manner that merely appears to be random. Makes no sense? Wrong!

A good random number generator needs to perform 3 functions:

(1) It must yield a set of uniformly distributed numbers over the specified range

(2) It must be able to yield all possible values (integers) in that range.

(3) It should be possible to generate the same sequence given the same initial parameters. This is to allow a program to be tested the same way each time it is run.

So, it would appear that a predictable generator is exactly what we want yet one which yields no apparent pattern of numbers. Such a generator exists in the Linear Congruential Method, first introduced in 1949. While this is not the only method, it is the most popular today, has been widely studied, and is very simple to use. Texas Instruments uses it in their programmable calculators.

Simply stated, the Linear Congruential Methods obey the equation:

$$X_{n+1} = (aX_n + c) \bmod m \quad \text{for } n \geqslant 0 \quad \text{where}$$

X_n is the starting value, X_{n+1} is the new value, a and c are chosen constants and m is the modulus

The best way to look at "mod m" is as the remainder left after division by m. For example: 7 mod 5 = 2, 15 mod 21 = 15, 3394 mod 1121 = 31. In the latter case we find the answer by dividing 3394 by 1121, dropping the digits to the left of the decimal and multiplying the fractional part by 1121. In BASIC the equation becomes:

```
((3394/1121)-INT(3394/1121))*1121
```

In the random number equation the values of a, c, and m can not be arbitrarily chosen and always produce a good generator. For example: try m=10, X_n=a=c=7. You should get the repeating sequence 7,6,9,0,7,6,9,0... This certainly isn't very random, is it? Why? The period, that is, the length of the repeating sequence, is too short. It may or may not be obvious also that the maximum period we can get will equal the modulus. But, as above, the modulus is 10 yet the period is only 4. I will not go into any detailed discussion of why this is so. The reader is referred to Donald Knuth's book The Art of Computer Programming, Volume 2, "Seminumerical Algorithms", pages 9-25 and following. I have the revised 1981 edition. It should be available in most college and technical libraries.

But now you must be asking what good all this is to you, the programmer? First, let me give you the rules for choosing the values of a, c, and m. (Knuth pp. 16, 170-171)

(a) The "seed" number X_n is arbitrary but must be less than m.

(b) m should be as large as possible but the computation of (aX+c)mod m MUST BE DONE EXACTLY with no round-off error.

(c) a should be between 0.01m and 0.99m

(d) c is arbitrary except that it must have no factor in common with m. Thus c=1 and c=a are acceptable.

(e) The sequence will have the full period m if and only if:
  i. c is relatively prime to m,(i.e. c and m have no common factors)
  ii. (a-1) is a multiple of P for every prime number P dividing m
  iii.(a-1) is a multiple of 4, if m is a multiple of 4.

(f) Some other rules apply to machine-language program generators. See Knuth's book.

Two examples: I have used a=c=1757, m=8192 and have verified that the period is indeed m. The generator works well if one does not need a longer period. I didn't for my application. Texas Instruments uses a=24298, c=99991, m=199017.

Now you may ask why even bother with your own generator when the CompuColor already has one built into it and a machine-language one at that? Well, the answer is: better control over the starting value. I use PEEK (33209) as my starting value. Since this puts the current value of the seconds of the real time clock in the equation, I can have 60 different sets of sequences (actually the same sequence begun at different points) and not have to worry about getting the same set of random numbers if I choose not to have them. Now I admit that Joe Charles' suggestion in "Basic Training For CompuColor Computers" is a nice one, but it does not perform the same function. All it does is push the sequence up to 60 numbers down the line. My suggestion could place the sequence just about anywhere if the generator is really random enough. A second advantage to my method is that you don't have to use CPU RESET to restore the generator. Just set $X_n=1$ in your program at the start of the generator. Remove it later or change the branch statement as necessary.

With all this under our belts you can now experiment. Devise you own generator, keeping the modulus small (say under 1000), and do some tests. Figure out the period. List the integers, values of X_{n+1}. Perhaps write a program to look at the distribution of numbers.

But my discussion would not be complete without one or two further comments. This method will normally yield INTEGERS. It is more often necessary to have values between 0 and 1 and adjust to a given range. I do this by considering the entire equation as it stands and not converting to mod m immediately. In other words, find the value (a*X+c)/m. This will be a number less than 1 if we drop off the digits left of the decimal. Note that the rightmost digits are less random, but this should not be any problem. But for now I will list my own program. Try it. Analyze it. Understand it. Then you will be ready for your own. Use it to generate various random values in any range you want. This is certainly necessary when designing games such as Dungeons and Dragons which use polyhedral dice (hence my article title). Random numbers are also needed for shuffling cards via computer (my next planned topic). So use these ideas in good health; and happy programming. Until next time.

```
100  RO=PEEK(33209)               140 RO=INT(C*8192)
110  A=1 : B=100                  150 R=INT((B-A+1)*RC+A)
120  C=(1757*RO+1757)/8192        160 Print R
130  C=C-INT(C)                   170 END
```

A and B are the limits of the random number, here 1 to 100. RC is the decimal portion between 0 and 1. R is the output random number. Note that statement 150 produces an R that <u>includes</u> the limits 1 and 100.

CompuPEEKing (#5) by Jim Minor

(Copyright 1981 by J.C. Minor)

The Array Space (cont.)

[Random File Buffers]

In the last installment we commenced looking at the array space and saw how numerical and string arrays were configured in that RAM space. This time we'll see how a another feature made available with the COMPUCOLOR BASIC makes use of that space. That feature is the random file capability and it uses the RAM array space by placing its file buffers there.

While PEEKing into this situation I obtained a better feel for how random files are implemented and how they should be used and I hope to pass some of that understanding on in this article by way of specific examples with random files as well as the particulars of the RAM buffers.

The first step in setting up a random file is its creation by using the FILE "N" ("Random File Creation") statement. When we execute the command

```
FILE "N","EXAMPL",16,50,5
```

we should interpret this as "Create a New file and call it 'EXAMPL', giving it a default type of .RND (other types could be used just as well); the file will consist of 16 records, each containing 50 bytes and we want to fetch them from the disk in 'physical blocks' of 5 records each (the 'blocking factor')." Upon execution of this command an entry will be made in the directory of the disk in the default drive. Assuming there is no other entry besides the 3-block directory itself, the disk directory display will look like this:

```
ATR  NAME TYPE VR SBLK SIZE LBC LADR SADR

 03 EXAMPL.RND;01 0003 0008  80 0010 0032
 01 <FREE SPACE>  000B 0185
```

Note that up to now there has been no change in the array RAM.

In order to properly understand this entry we should take a closer look at the consequences of what we have asked for, keeping in mind the distinction between a physical block (consisting of 5, 50-byte records as per our instructions) and a disk block (also referred to as a disk "sector", they are 128 bytes long with 400 of them on a disk as per COMPUCOLOR's FCS). On our disk (which has disk blocks 0-2 devoted to the directory) we have specified that our random files are to be laid out as shown in Figure 1. Note that records 3, 8, and 13 span (i.e. cross) disk blocks but that no record is allowed to span any physical blocks.

```
           :<----Disk Block #3 (128 bytes)--->:     :<----Disk Block #4 (128 bytes)--->:

Physical  ||===================================||   ||===================================||
Block #1: || Record #1   : Record #2   : Record ||   ||#3   : Record #4   : Record #5   :xxx||
          ||===================================||   ||===================================||

           :<-50 bytes->:<-50 bytes->:<--- 50 bytes --->:<-50 bytes->:<-50 bytes->:<6>:



           :<----Disk Block #5 (128 bytes)--->:     :<----Disk Block #6 (128 bytes)--->:

Physical  ||===================================||   ||===================================||
Block #2: || Record #6   : Record #7   : Record ||   ||#8   : Record #9   : Record #10  :xxx||
          ||===================================||   ||===================================||

           :<-50 bytes->:<-50 bytes->:<--- 50 bytes --->:<-50 bytes->:<-50 bytes->:<6>:



           :<----Disk Block #7 (128 bytes)--->:     :<----Disk Block #8 (128 bytes)--->:

Physical  ||===================================||   ||===================================||
Block #3: || Record #11 : Record #12 : Record ||   ||#13 : Record #14 : Record #15 :xxx||
          ||===================================||   ||===================================||

           :<-50 bytes->:<-50 bytes->:<--- 50 bytes --->:<-50 bytes->:<-50 bytes->:<6>:



           :<----Disk Block #9 (128 bytes)--->:     :<----Disk Block #A (128 bytes)--->:

Physical  ||===================================||   ||===================================||
Block #4: || Record #16 :xxxxxxxxxxxxxxxxxxxxx||   ||xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx||
          ||===================================||   ||===================================||

           :<-50 bytes->:<-----78 bytes------>:     :<----------128 bytes------------->:
```

FIGURE 1. DISK ALLOCATION AS A RESULT OF THE COMMAND: FILE "N","EXAMPL",16,50,5.
========= Here the word "RECORD" is equivalent to "Logical Record".

Reviewing the file directory, we see that the starting block (SBLK) and SIZE information is just as we'd expect: the random file EXAMPL starts at disk block #3 and runs for 8 consecutive blocks. Note that most of disk block #9 and all of disk block #A are wasted as a consequence of OUR selectiion via the FILE "N" statement. The Last Byte Count (LBC) is 128.(80H) which just says that the last disk block (#A) is completely reserved (even though it won't be used). The "Load Address" (LADR) is 16.(0010H) which for a random file is interpreted as the number of records and finally the "Start Address" (SADR) is 50.(0032H) which for a random file is interpreted as the number of bytes in a record. The last bit of information that we need, the blocking factor (=5 in our example,) is in the disk directory but is not displayed when you call "FCS>DIR". It is stored in the directory byte #43 which is labeled "RESERVED" by Dennis Martin on pages 9-10 of the May 1980 (v.3,#5) issue of COLORCUE.

Up to now all we have done is reserve space on the disk and recorded in the directory how that space is to be allocated ("parsed"). To make use of this file the next step is to PREPARE buffers in RAM which can be used as virtual disk space (or we can think of it as disk space that becomes virtual RAM). This preparation is all done with the File "R" (Random File Open) command. Continuing with our example, we now enter

```
FILE "R",27,"EXAMPL",2
```

which should be interpreted as "Reserve buffers for a file to which we shall give the RAM code number '27' but which is called 'EXAMPL' on the disk; reserve space for 2 physical blocks". The FILE "R" statement could have additional arguments which override the record-defining parameters given in the FILE "N" statement but we won't be employing that option here.

The FILE "R" statement does affect array RAM and we now launch into a description of what it does here. Starting with the position pointed to by the End of Array pointer (EOA or EA = 256*PEEK(32987)+PEEK(32986)), the BASIC interpreter fills 34.(22H) bytes of RAM with bookkeeping information necessary to implement the random file system for that particular file. This bookkeeping space (which I shall call "the file header") is followed by the virtual image space for the file's physical blocks. Each physical block space in RAM (or "buffer") is preceeded by 4 bytes of bookkeeping space for that particular physical block (I term these 4 bytes "the buffer header"). In our example the FILE "R" statement asked that space be reserved for two such physical blocks (i.e. two buffers) and the FILE "N" statement had previously defined a physical block (=1 buffer) as 5, 50-byte records. In reserving this space, the End of Arrays pointer (i.e. the contents of 32986.(80DAH) and 32987.(80DBH)) is moved up the appropriate distance [Increase = 34.+2*(4.+2*128.)].

In summary, the RAM space for our example is partitioned as follows:

File bytes 0->33.	FILE HEADER. Information pertaining to this particular file. File Headers are always 34. bytes long.
File bytes 34.->37.	BUFFER "A" HEADER. Information pertaining to the contents of the first buffer.
File bytes 38.->293.	BUFFER "A". Room for 1 physical block (5, 50-byte records) of information.
File bytes 294.->297.	BUFFER "B" HEADER. Information pertaining to the contents of the second buffer.
File bytes 298.->553.	BUFFER "B". Room for 1 physical block (5, 50-byte records) of information.
Byte 554.	The byte which is now pointed at by EOA. If more (different) random buffers or new arrays are created then they would start here (and EOA moved accordingly).

Up to now no information has been moved into BUFFER "A" or BUFFER "B" or their 4-byte headers. Before issuing commands to use these buffers, we first call attention to TABLE 1.(next page) which describes the contents of bytes 0->33. in greater detail. I should remind the reader that my methods for determining the use of these bytes are rather primitive. I make changes in my commands and then PEEK to see what effect they have on particular RAM locations. Answers obtained by such correlations are always suspect but for the most part I believe they're indicative of RAM usage even if not always 100% accurate in describing cases I haven't tested.

We start loading information into the fle by using the PUT command:

```
PUT 27,7,3;"MAY 1981"[8]
```

which should be read as "Put the expression(s) following the semicolon into the file with filecode '27' (which the FILE "R" statement equated with 'EXAMPL.RND'); the expression(s) goes into logical record #7 starting with the 3rd byte". Upon entering this command, physical block #2 (which contains record #7) will be loaded into BUFFER "A" in the array RAM. Furthermore, the expression "MAY 1981" will be written to the appropriate record #7 image in RAM and the following header changes will be made:

- Bytes 9 and 10 of the file header will be changed from 0000H to 0007H reflecting the fact that record 7 was the most recently changed.
- Bytes 11 and 12 of the file header will be changed from 0000H to 0001H, reflecting the fact that we have exercised one PUT (or GET) to (from) this file.

(continued following TABLE 1.)

Bytes 0->3 are used in the same manner as the first four bytes in numerical and string arrays:

- Bytes 0 and 1 give the file code name. In our example the code name was "27" so byte 1 contains 27. Byte 2 is always 0 in a random file and this serves to distinguish it from arrays. [=27.(001BH)]
- Bytes 2 and 3 tell how many bytes lie BETWEEN byte 4 and the start of the next array or file space. Thus if this is not the file/array we're looking for, skip over (256*byte4 + byte3) bytes to get to the next name. [=550.(0226H)]
- Byte 4 always seems to be zero.

Bytes 5->8 are useful in parsing the buffer space:

- Byte 5 gives the number of buffers in RAM. [=2]
- Byte 6 gives the number of disk blocks per buffer. [=2]
- Bytes 7 and 8 give the number of bytes per buffer. [=256.(0100H)]

Bytes 9->12 contain information that is useful when making sequential PUT or GET statements:

- Bytes 9 and 10 specify the last record that was accessed by a PUT or GET. [=0 until a PUT/GET statement is used]
- Bytes 11 and 12 specify the number of PUTS/GETS to this file. This proves useful when we need to know the least recently used file. [=0 until a PUT/GET statement is used]

Bytes 13-25 are copied from the disk directory:

- Bytes 13 and 14 are the last two letters of the file type. [=(R)"ND"]. Probably the directory information was copied into bytes 4->25 and then subsequently partially written over, all as part of the FILE "R" statement.
- Byte 15 is the file version number. [=01]
- Bytes 16 and 17 are the file SBLK. [=0003H]
- Bytes 18 and 19 are the file SIZE. [=0008H]
- Byte 20 is the Last Block Count. [=128.(80H)]
- Bytes 21 and 22 are the LADR (i.e. # records). [=16.(0010H)]
- Bytes 23 and 24 are the SADR (i.e. # bytes/record). [=50.(0032H)]
- Byte 25 is the number of records/block. [=5]

Bytes 26->27 contain information as to where this file is REFERENCED within the directory:

- Byte 26 is the reference block number within the directory (00 = 1st block). [=00H]
- Byte 27 is the reference position within the directory block (06 = 1st, 01 = last). [=05H]

Bytes 28->33 contain miscellaneous information:

- Bytes 28 and 29 contain the SIZE of the last file created (=FF00H if none created this session). [=0008H]
- Bytes 30 and 31 contain the address of the reference to the FCS disk handler. [=368EH for v6.78]
- Byte 32 seems to indicate if ANY changes have been made to the buffers. 00 = no change, 0FDH = change.
- Byte 33 always seems to be zero.

TABLE 1. FILE HEADER BYTE ALLOCATION. Quantities in square brackets, [ ], are values obtained for the example given in the text.

The four buffer header bytes are used as follows:

> The first two bytes identify the first record of the physical block in that buffer.
>
> The third byte gives the PUT/GET sequence number for this buffer (just a count). This is used in determining which buffer was LEAST recently accessed.
>
> The fourth byte is a flag (bit 7 ON) to indicate that a change has been made in the contents of the buffer which has not been recorded on the disk.

Thus for our example, the four BUFFER "A" bytes (34.->37.) are changed as follows:

> Bytes 34 and 35 give 0006H since this physical block #2 (which contains record #7) STARTS with record #6.
>
> Byte 36 is 01H since this is the first PUT/GET to this buffer.
>
> Byte 37 is 80H since this change has not been transferred to the disk.

The Put command also causes the information in disk block (or sector) numbers 5 and 6 (i.e. physical block #2) to be copied into the 256. bytes (38.->293.) which represent BUFFER "A". In our example the "information" in physical block #2 is whatever was last left on disk blocks 5 and 6 before we appropriated them for our random files. In any event the system dutifully copies it into the buffer and it is up to us to make intelligent use of it (i.e. don't try to GET anything out of it before we first PUT something in). Finally, the PUT command places the expression, "MAY 1981", into logical record #7 starting at bit 3 and, if you're really interested, this information occupies file bytes 90. to 97..

If we execute a 2nd PUT command thusly,

```
PUT 27,15,2;"JUNE 1981"[9]
```

then in the file header bytes 9 and 10. are changed to represent 15.(000FH) (i.e. the record being changed) and bytes 11. and 12. are increased to bring the count of PUT/GETs to this file to 2.(0002H). Furthermore BUFFER "B" is filled with the contents of disk blocks 7 and 8 (i.e. physical block #3) and the ASCII string for "JUNE 1981" is written into the appropriate position of the logical record #15 image starting with byte 2 (i.e. file bytes 499. to 507.). The BUFFER "B" HEADER has its first two bytes (i.e. file bytes 294. and 295.) represent 11.(000BH), the third byte is 2 and the final byte is, of course, 128.(80H).

If we execute a 3rd PUT, e.g.

```
PUT 27,4,2;"JULY 1981"[9]
```

then the power of the random fles comes into play in that the system sees (via file bytes 11, 12, 36, and 296) that BUFFER "A" was less recently used (and for more complicated problems a possibly less active file) than BUFFER "B" and so the contents of BUFFER "A" will be written back to the disk and Physical Block #1 (containing record #4) will be read into that buffer.

If we next execute a GET, e.g.

```
GET 27,7,3;A$[8]
```

then BUFFER "B" will be recognized as the least recently used so its contents (Physical Block #3) will be written back to the disk and Physical Block #2 (which contains logical record #7) will be written into BUFFER "B". There's an added twist here in that room must be made for the string variable, A$; thus the entire array space is shifted up in RAM 6 bytes (cf. CompuPEEKing #3). This shift is of course transparent to the BASIC user. We also note that the fourth byte in the BUFFER "B" HEADER is 00H signifying that there is no information in that buffer that is not already on the disk. If we execute a PRINT A$ now, we get our original entry into logical record #7, i.e. "MAY 1981".

If we execute a FILE "D" (Dump File Buffers) for this file, i.e.

```
FILE "D",27
```

then BUFFER "A", which has an unrecorded change, will be written to the disk; byte 32. in the file header will be reset to 00; and similarly the 4th bytes in the two buffer headers will each be set to zero.

Finally, if we execute a FILE "C" (Random File Close), i.e.

```
FILE "C",27
```

then any buffer that had not had its changes transferred to the disk (in the present sequence there won't be any since the previous FILE "D" took care of that) will be recorded and then any arrays or other random file spaces higher in memory will be shifted down and EOA moved accordingly. Its function is basically just the opposite of the FILE "R" command.

This concludes our investigation into the "array" space.

Coming up next - The structure of numbers in BASIC.

ADDENDA

Since I've got a little space left on this page I thought I'd share a recent discovey.

A BASIC program usually starts at location 33434.(829AH); however for proper execution the byte before it (i.e. 33433.(8299H)) must be zero. If this location is not zero then the program will LIST properly and a GOTO any line other than the first will give proper execution but a RUN or a GOTO the first line will result in a SN (syntax) ERROR. This situation is frequently encountered if you've been running a machine language program and then load a BASIC program without first pressing ESC W. Solutions are to POKE 33433,0, or GOTO the second line number (assuming the first was a REM and you don't need to initialize variables), or press ESC W and reload the program. The later solution is probaly the safest since there may be other information in bytes 8200H->8298H which you may need to use later but is not properly implemented.

ROCHESTER NY 146
PM
22 [illegible]
1991

PAUL McCARRON
PO BOX 108
BELGRADE LAKES, MA 04918

Bob Nickles.